Repositories & CLIs – “Code as cattle not pets”

Presenter: Brian Greene | CTO NeuronSphere

Is each source repository in your org its own unique pet?

Does each one require special incantations to build or test?

Is CI/CD consistent and transparent?

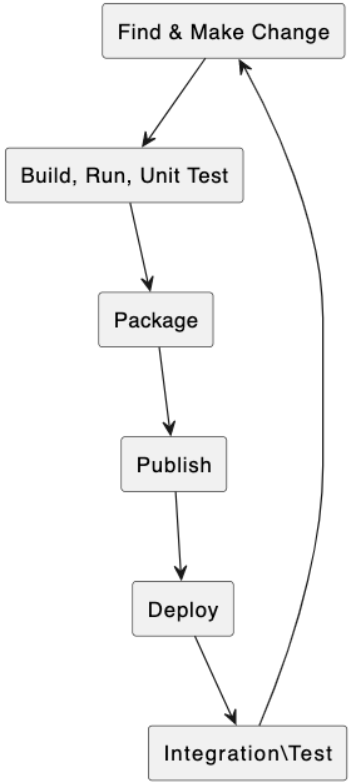
Accounting for cross-technology & lifecycle repository management demands and how to design an extensible CLI framework to streamline developer workflows

Goals (want)

1. Allow developers to think less & go faster
2. Make it easier to break Conway's Law
3. Produce higher-quality software

** Assumes we are doing N of things*

(what)



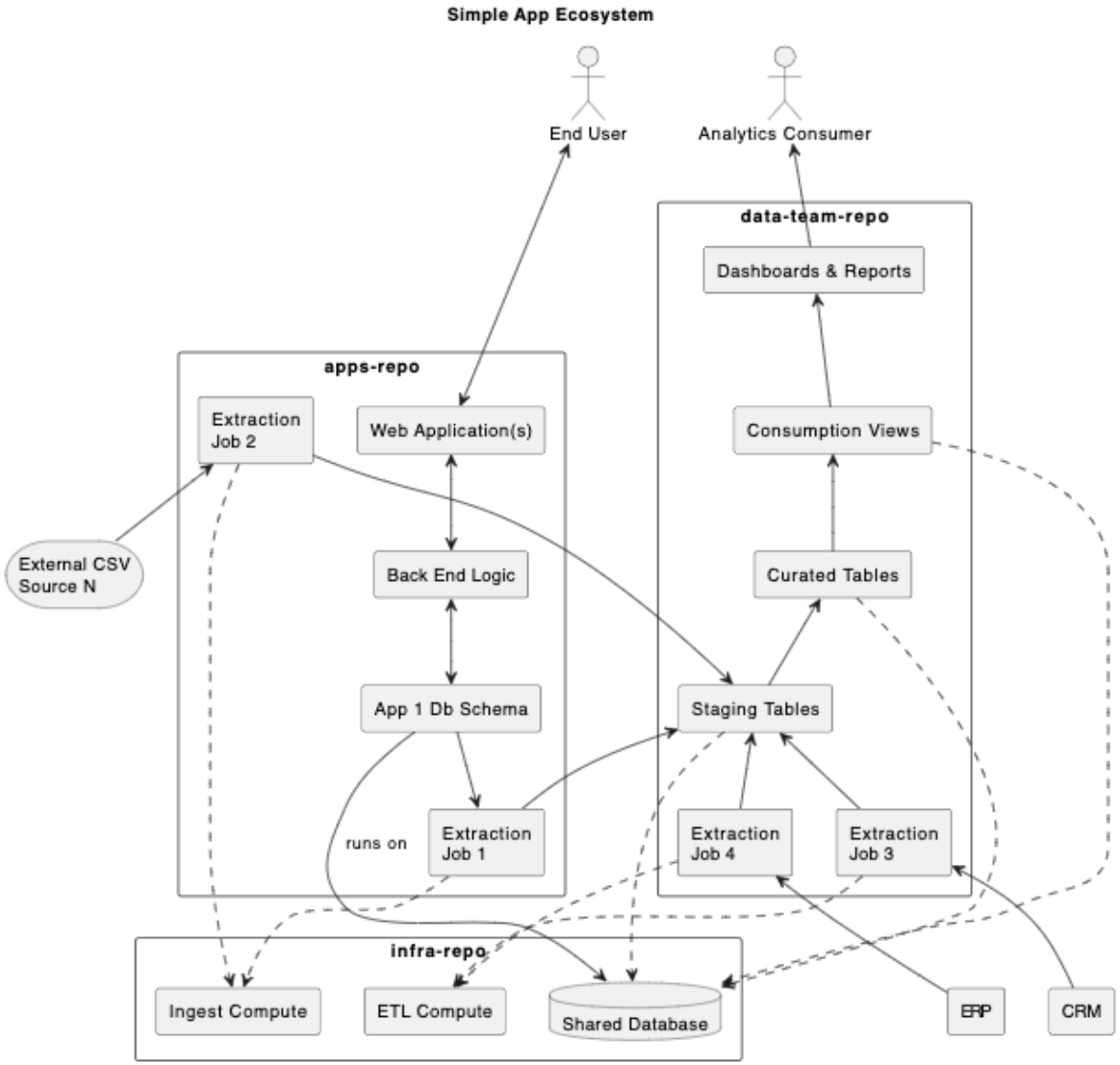
Strategy (how)

1. Apply a component-based development approach, anchored in a poly-repo code management strategy:
   - Everything is Code
   - Universal Versioning
   - Dependency Management
   - Packaging Consistency
   - Artifact Output/Distribution
2. Invert and unify CLIs to manage #1
   - If everything is code, and the code is laid out consistently, then the repo is the parameter!

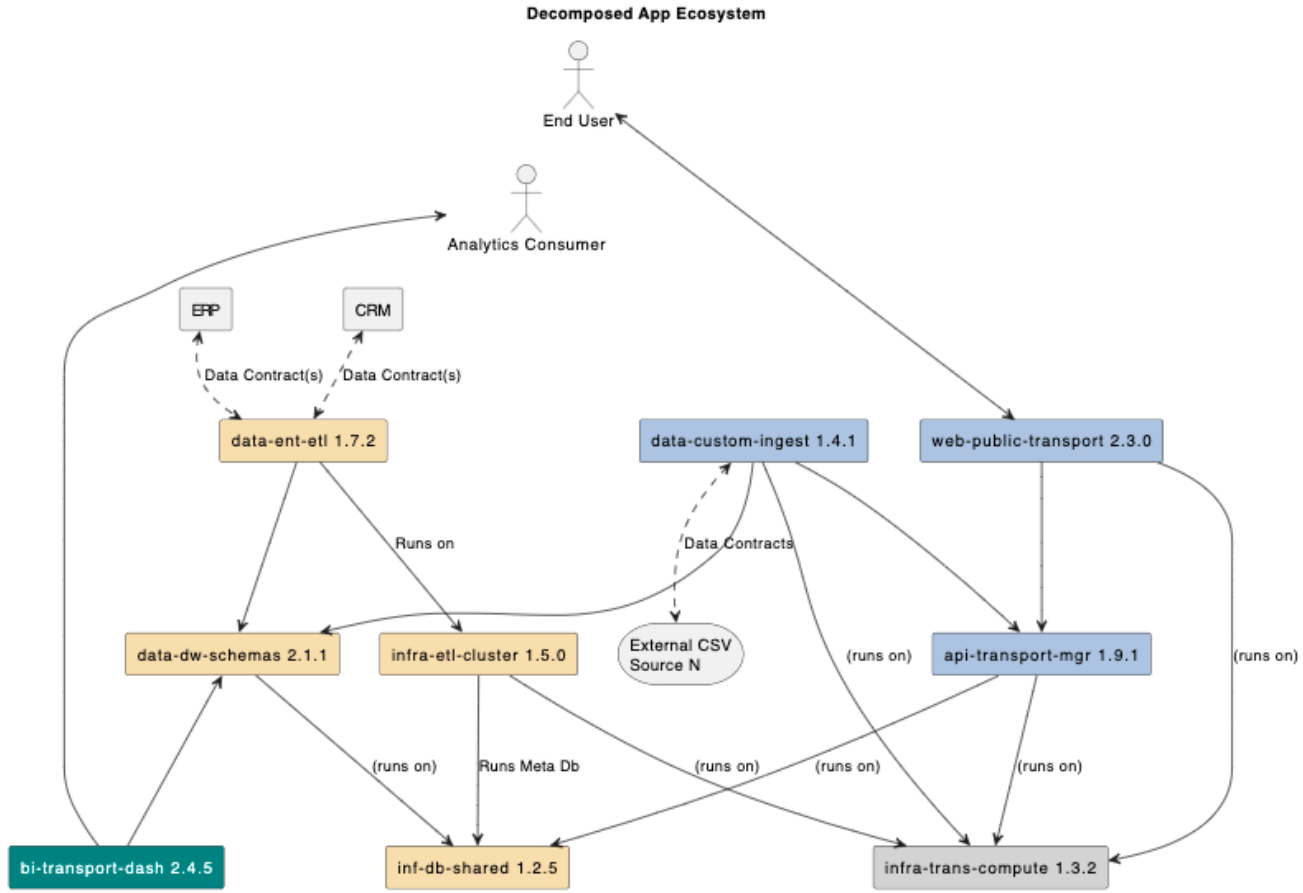
Conway's Law: Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure.

Anti-Conway Software delivery starts with many small repositories focused on artifact interaction patterns with orthogonal tooling to manage them*

Example Before



After (with revs)



Code Repositories (as) ~Architectural Quantum

”Everything is code in git” is a great baseline
- but not nearly enough

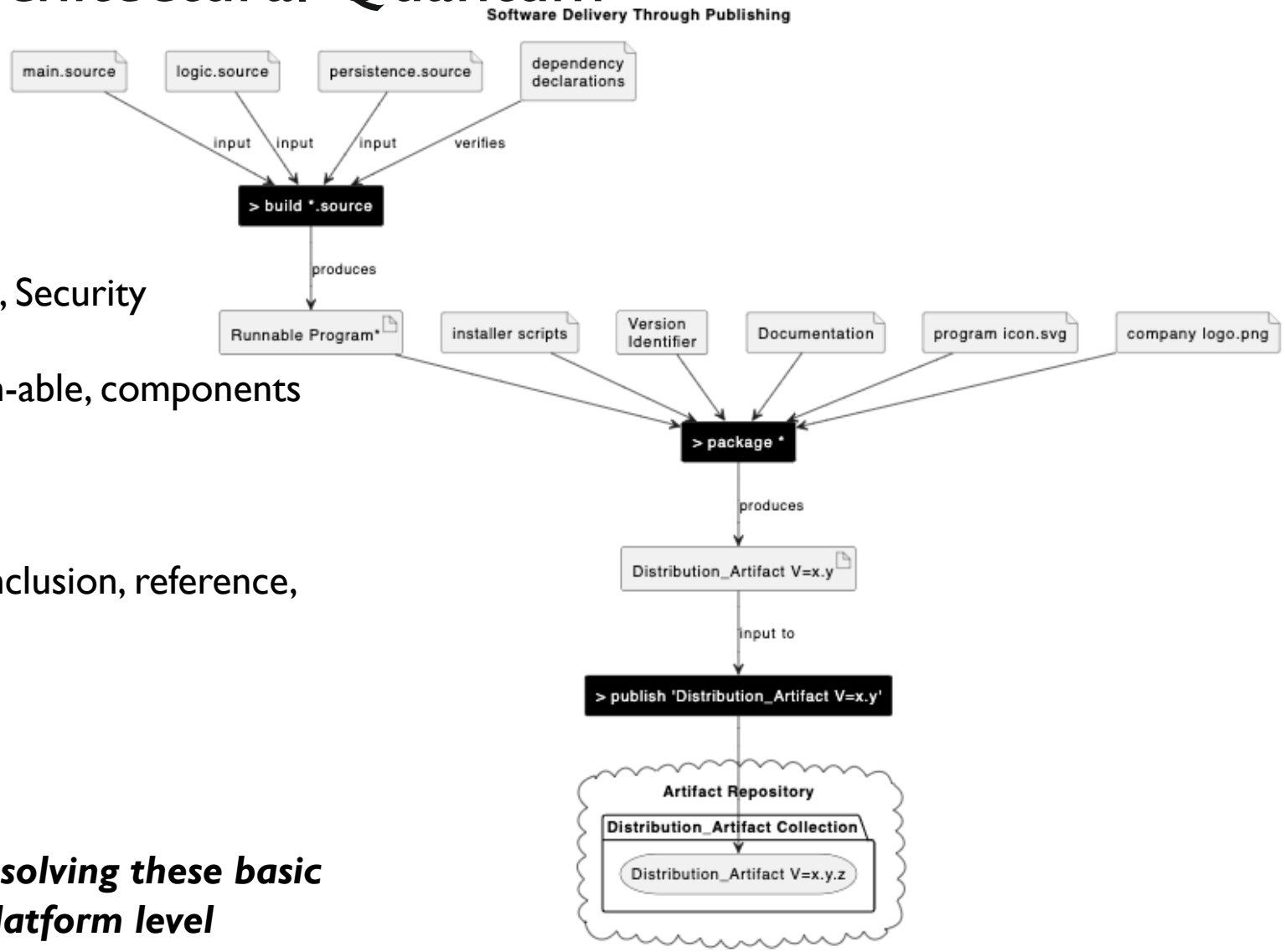
We all have a diverse architecture that's growing
- Software, Data models, Infrastructure, Analytics, Security

It exists as a collection of self-referencing, version-able, components representable as Artifacts
- (whether you manage it that way or not today)

All technologies provide mechanisms for reuse, inclusion, reference, leading to the core capabilities of:

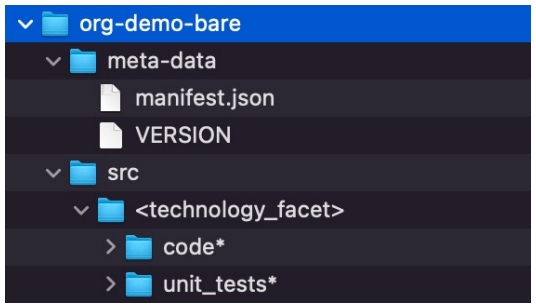
- Versioning
- Packaging
- Dependencies
- Artifacts (Distribution)

Many challenges are easier after consistently solving these basic problems at a cross-team, cross-technology, platform level



What's a basic standard repo look like?

- Very mvp



The manifest is a contract:

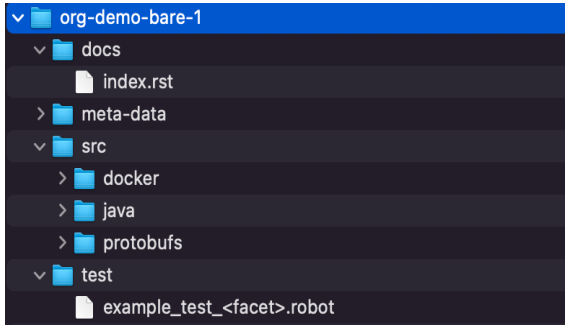
- with other repos
- with platform tooling
- with a deployment environment

VERSION is a touchstone

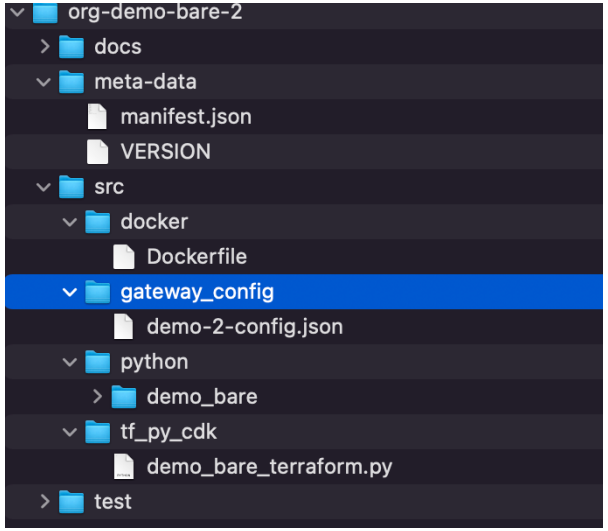
The layout is a contract:

- Orthogonal platform tooling is easy to develop with a consistent repository layout

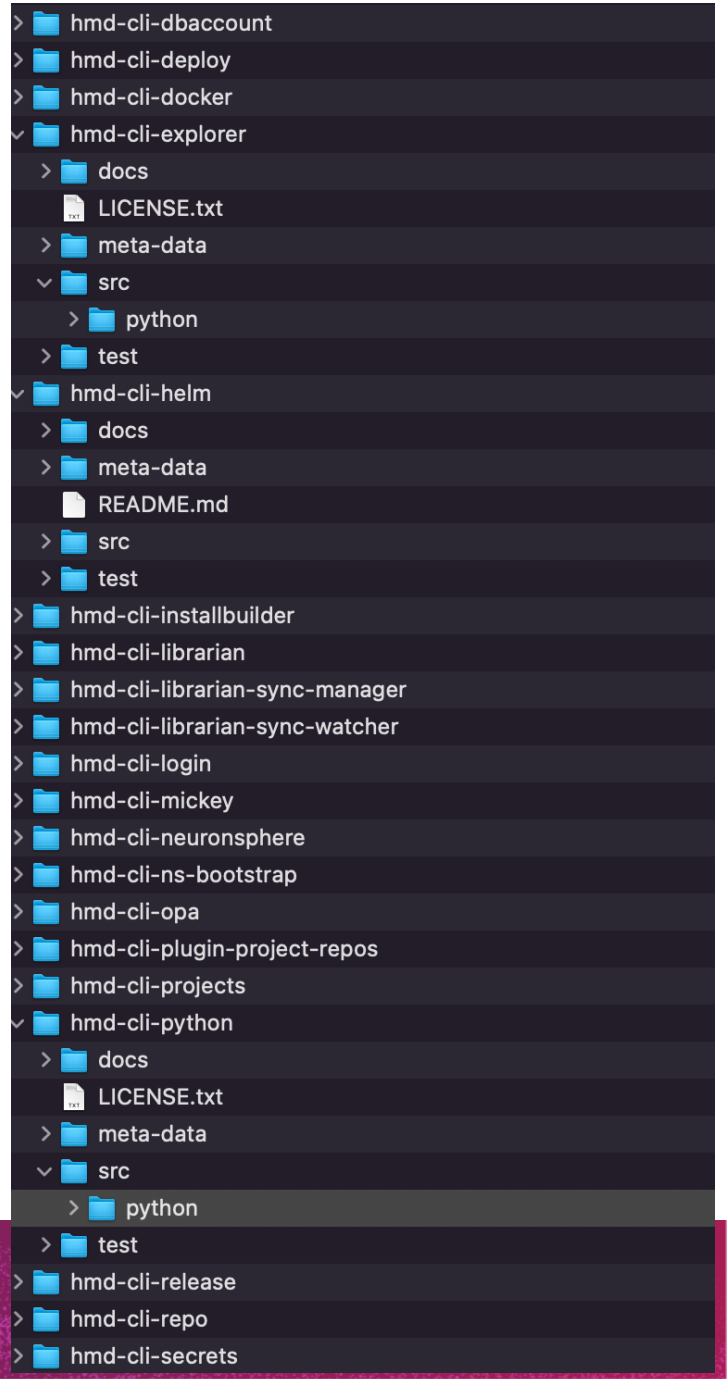
- Docs and Integration tests, better



- Example of 'facets' (rather than by tech)



- An example of a collection of repos



Inverted, platform-supported CLIs for speed and governance

"How to build this repository?" – a question no dev in a platform ecosystem should ever have to utter.
Repositories with a standardized layout and interaction models gain an exponential lift by having CLIs that work with a *type* of repository or perform actions on all *standard* repositories to the same effect.
Example commands:

```
demo_code/repo_x/:> build
demo_code/repo_x/:> <tech_x> package
demo_code/repo_x/:> <tech_y> publish
demo_code/repo_x/:> docs
```

A "good CLI" in this strategy:

- wraps a technology or facet into a consistent CLI command & lifecycle interface
- interacts with the manifest for per-repo overrides & config
- Consistently interacts with the standard layout wrt working space

CLIs are governed by the platform team, built by technology practitioners - harmonizing behaviors while encoding standards and defaults *orthogonally.*

A standard layout allows the platform team to determine where CLIs are missing & where common options need to be reconsidered.

This makes CI much easier to reason about, as the CLIs that devs use locally to build a repo will build it anywhere

Org X <technology_facet/> list	Template(s)	Build (&ut)	Package	Publish	Deploy	Int-Test*
Java (lib/app)	2	Y	Y	Y	Y	na
Protobuf defs	1	Y	*	*	na	na
Docker (general)	1	Y	na	Y	na	L/C
DB Migrations? (e.g.: /src/alembic)	1	Y	Y	Y	na	C
SQL Scripts (Snowflake T & V)	2	na	zip		Y	C
ETL Configs	2	na	zip			C?
Docs!	N*	Y	varies	Y*	na	L
…						

How to start now (new repos)

1. Decide on a few facets and technologies you need N of, libs in your default lang is a good first choice
2. Docker images a decent second
3. Docs and diagrams as code a reasonable third
4. A couple layered templates
5. Probably some Infrastructure as Code modules - TF module, SLS base pieces, or Pulimi all decent options

How to scale & impact legacy code

1. Then you scale the practice by iteratively applying it to existing repos, chopping them up as reasonable and useful
   - Find a place in your architecture that causes release management challenges, or is deeply challenged by Conway's law, and break things up
2. Extract and version a data model so its code generation generates external libs rather than in-build-situ "temp" libs
3. Look for subsections you wish you had better testing for, and use extraction to a repo as a lever
4. Infra-instances & multi-instance configs are another place where separation of concerns by repo facet can be operationally impactful (deploy app v=x with config v=y to target=z)

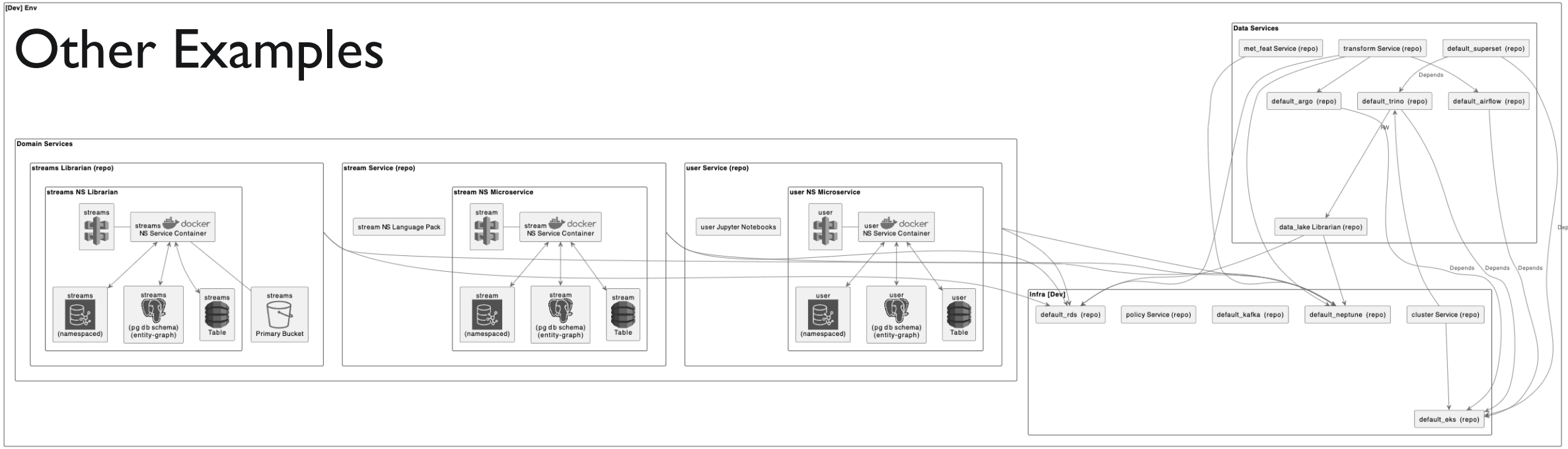
Common Responses

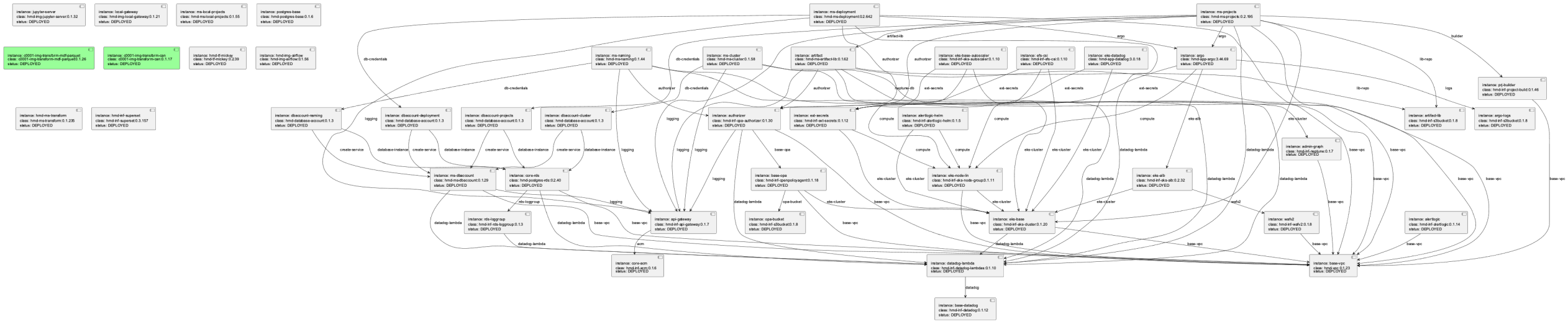
1. <foo/> uses a monorepo
2. I don't like microservices
3. Generated repos get stale
4. Dev teams should have the flexibility to define their own repo layouts
5. You need tooling for this
6. Same problems in a monorepo/you can do that in a monorepo
7. This is painful with gitflow

Common Side-Effects

1. ‘Architecting with repos’ will naturally produce a more modular, maintainable, and testable ecosystem of software components, regardless of target deployment architecture
2. Easy to create mechanisms to express runtime dependencies as abstractions for service-locator style interactions
3. It’s easier to perform meta-analysis on the codebase (e.g.: do <x> to all the python libs is much easier to reason about, as is the question “how much documentation do we have for component y?”)
4. Larger experiments with design happen faster with the reusable components and architecture patterns – it is easier to propose new things and see how they will fit (this works quite well for Infra as Code)
5. Developers can move across larger expanses of code more quickly, focusing on learning a new tech or subsection rather than the ideas of each repository designer or vendor
6. Provides a clear path for how to introduce a new kind of technology into the ecosystem as a one-off and then with developed standards and harmonized tooling – a happy path for the platform team!
7. Standardized wrapper CLIs also allow instrumentation and governance around tool use in the software supply chain
8. Pairs well with Trunk Based Development and aggressive and automated CI/CDD ecosystems

Other Examples





Platform engineering for [data]